

# PATENT ABSTRACTS OF JAPAN

(11)Publication number : 05-204675

(43)Date of publication of application : 13.08.1993

(51)Int.CI.

G06F 9/46

(21)Application number : 04-015120 (71)Applicant : TOSHIBA CORP

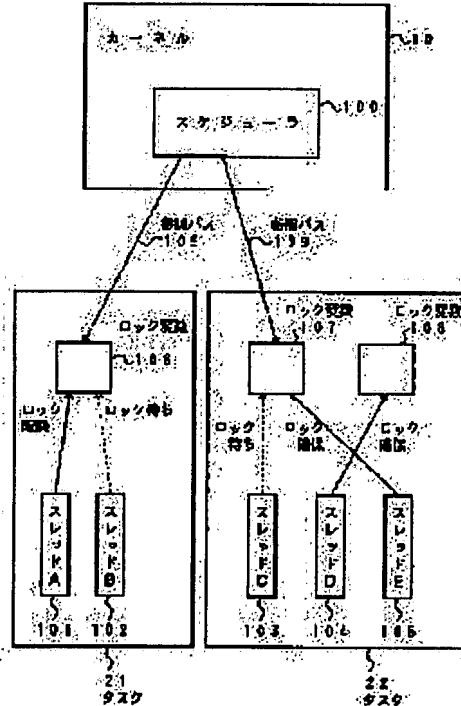
(22)Date of filing : 30.01.1992 (72)Inventor : MIZUNO SATOSHI

## (54) SCHEDULING SYSTEM

### (57)Abstract:

**PURPOSE:** To improve the using efficiency of a CPU by executing scheduling based upon whether a thread or a process is in a lock queuing state or not.

**CONSTITUTION:** At the time of scheduling, a scheduler 100 refers to a lock variable 106 corresponding to an execution candidate thread 101 when necessary and checks whether the thread can be instantaneously locked or not to use the checked result for the reference to judge the allocation of the CPU. Since a thread which is not in a lock queuing state, i.e., a thread waiting for an unlocked variable, or a thread in a locked state can be selected and executed with priority, the useless consumption of CPU time due to a spin loop can be reduced and efficient processing can be attained.



## LEGAL STATUS

[Date of request for examination] 19.09.1996

[Date of sending the examiner's decision of rejection] 28.07.1998

[Kind of final disposal of application other than the examiner's decision of rejection or

application converted registration]

[Date of final disposal for application]

[Patent number] 2866241

[Date of registration] 18.12.1998

[Number of appeal against examiner's  
decision of rejection] 10-013148

[Date of requesting appeal against  
examiner's decision of rejection] 27.08.1998

[Date of extinction of right]

(19)日本国特許庁 (JP)

(12) 公開特許公報 (A)

(11)特許出願公開番号

特開平5-204675

(43)公開日 平成5年(1993)8月13日

(51)Int.Cl.<sup>5</sup>

G 0 6 F 9/46

識別記号 庁内整理番号

3 4 0 F 8120-5B

F I

技術表示箇所

審査請求 未請求 請求項の数2(全17頁)

(21)出願番号 特願平4-15120

(22)出願日 平成4年(1992)1月30日

(71)出願人 000003078

株式会社東芝

神奈川県川崎市幸区堀川町72番地

(72)発明者 水野 聰

神奈川県川崎市幸区小向東芝町1番地 株

式会社東芝総合研究所内

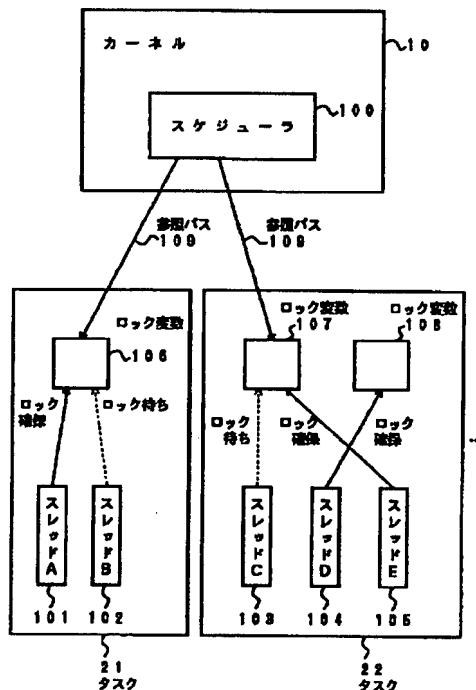
(74)代理人 弁理士 鈴江 武彦

(54)【発明の名称】 スケジューリング方式

(57)【要約】

【目的】スレッドまたはプロセスがロック待ち状態か否かに基づいてスケジューリングを実行できるようにし、CPUの利用効率の向上を図る。

【構成】スケジューラ100がスケジューリングの際に、必要に応じて実行候補のスレッド101に対応するロック変数106を参照し、そのスレッド101がすぐにロックを確保できるかどうかを調べ、CPUを割り当てるか否かの判断の参考とする。この結果、ロック待ち状態にないスレッド、つまり解除されたロック変数を待っていたスレッド、またはロック状態のスレッド、を優先的に選んで実行できるので、スピルループによるCPUタイムの無駄な消費が少くなり、効率よい処理が実現できる。



1

## 【特許請求の範囲】

【請求項1】複数のプログラム実行単位間の相互排除を共有変数を用いて実現するコンピュータシステムにおいて、

前記複数のプログラム実行単位それぞれがロック待ち状態にあるか否かを管理する手段と、

ロック待ち状態でないプログラム実行単位に対して優先的にCPUが割り当てられるように前記複数のプログラム実行単位の処理順を決定するスケジューリング手段とを具備することを特徴とするコンピュータシステムのスケジューリング方式。

【請求項2】複数のスレッドあるいはプロセス間の相互排除を共有変数を用いて実現するコンピュータシステムにおいて、

複数のスレッドあるいはプロセスの処理順を一定時間間隔で順次決定するスケジューリング手段を具備し、

このスケジューリング手段は、

実行候補の複数のスレッドあるいはプロセスの中から1つのスレッドあるいはプロセスを実行許可判定対象として選択し、その選択したスレッドあるいはプロセスに対応する共有変数を参照する手段と、

この参照結果に基づいて前記選択したスレッドあるいはプロセスがロック待ち状態にあるか否かを判断し、ロック待ち状態でない際に前記選択したスレッドあるいはプロセスにCPUを割り当てる手段と、

前記選択したスレッドあるいはプロセスがロック待ち状態の際に、前記実行候補の複数のスレッドあるいはプロセスの中から別の1つのスレッドあるいはプロセスを前記実行許可判定対象のスレッドあるいはプロセスとして選択する手段とを含んでいることを特徴とするコンピュータシステムのスケジューリング方式。

## 【発明の詳細な説明】

## 【0001】

【産業上の利用分野】この発明はコンピュータシステムにおけるスケジューリング方式に関し、特に複数のスレッドあるいはプロセス間の相互排除を共有変数を用いて実現するコンピュータシステムにおいてそれら複数のスレッドあるいはプロセスの処理順を決定するためのスケジューリング方式に関する。

## 【0002】

【従来の技術】従来、コンピュータシステムにおいて、オペレーティングシステムによって行われるスケジューリングは、各プロセス、各ユーザ間に、なるべく公平にCPUタイムを割り振るような方針で行われていた。しかし、近年、マルチプロセッサシステムが開発され、スレッドプログラミングを代表とする並行/並列プログラミングが大きく取り上げられるに伴い、従来のスケジューリング方法では十分良い効率が得られないという問題を生じ、新しいスケジューリング手法の開発が望まれている。

2

【0003】まず、スレッドについて説明する。図10には従来の典型的なオペレーティングシステム（例えば、Unix）におけるプロセスのメモリ空間の構成が示されている。なお、図中の黒丸印は実行単位（CPU割り当ての単位）を示す。

【0004】この図10に示すように、各プロセスは個別のテキスト、データ、スタック空間を有し、互いに参照し合うことはできない。プロセス間の通信、同期はシステムコールを使わざるを得ず、オーバーヘッドが大きい。この構成だと複数のプロセッサが協調して効率よく1つの処理を行うようなプログラミングは困難である。そこで考え出されたのがスレッドプログラミングである。図11にスレッドのメモリ空間構成を示す。

【0005】スレッドは実行の単位であるが、複数のスレッド間で同一のテキスト空間、データ空間、スタック空間を共有することができる。このため、データ空間に共有している変数を使うことにより、システムコールを使わずに高速なスレッド間の通信および同期を実現でき、マルチプロセッサシステムにおけるプログラミングに適している。なお、同一空間を共有するスレッドをまとめてタスクと呼ぶ。

【0006】また、同様な考え方で、図12に示されているように、従来のプロセスに共有メモリを導入することも行われている。共有メモリは複数のプロセス間で共有できる特別なデータ空間であり、やはりマルチプロセッサシステムにおける高速なプロセス間通信、同期に利用することができる。以降、スレッドに関して話を進めるが、プロセス間の通信/同期に共有メモリを使う場合についてもスレッドの場合と同様である。

【0007】一般的なスレッドプログラミングでは、複数のスレッド間で共有する変数をスレッド間の同期（相互排除）に用いる。以降、この様な変数を共有変数、あるいは、ロック変数と呼ぶ。

【0008】図13にその一例を示す。この図13では、タスク1のスレッド1、2、3、4が、ロック変数Sを用いて、スレッド間で排他的に特定の処理を行っている。ロック変数Sの値が1の時は、ある1つのスレッド（この図ではスレッド3）が排他的な処理を実行中であること（ロック状態）を示し、他のスレッド1、2、4はその処理を実行することができない。

【0009】図14に、そのような排他制御を実現する処理の流れをフローにして示す。図中、ステップS1、S2、S3はロックを確保するときの処理の流れであり、ステップS4はロックを解除するときの処理の流れである。

【0010】ロック確保時のロック変数Sの参照、更新は、テストアンドセット命令（test & set）のように参照と変更が不可分に行われるプロセッサ命令で行う必要がある。

50 【0011】テストアンドセット命令は、基本的には不

可分の単一機会命令として、指定されたロック変数から値“0”を読み出した後、“1”の値をロック変数に書き戻すものであり、最初に“0”（ロックがオフ）を受け取るスレッドだけがロックを確保して処理を実行でき、それ以外のスレッドはすべて“1”（ロックがオン）を受け取ることにより実行を待たれる。この場合、“1”を受け取った各スレッドは、ロック変数が“0”になるまで繰り返しテストを行い、ロック解除されまで待つ。この状態は、繁忙待機 (busy wait)、またはスピルループと称されている。

【0012】このようにロック変数を用いて相互排除を実現する方法はプログラミングが容易で、かつ確実にスレッドの同期を取ることができる。しかし、複数のスレッドが頻繁にロック変数による同期を行う際には実行効率が悪化する場合がある。例えば、あるスレッドがロックの解除を待つスピルループしている途中で実行が止まっていた（CPUが取り上げられていた）状態であり、かつそのスレッドが次の実行の候補に選ばれたときにロックがまだ解除されていないような場合、そのスレッドを実行しても無駄にスピルループをするだけで実質的に処理は進まない。このようなロック待ちでスピルループしているスレッドを頻繁に選択し、実行するほどシステム全体の効率は悪化する。

【0013】通常のオペレーティングシステムは、実行中のスレッドを決められた時間（単位実行時間、タイムクォンタムということがある）連続して実行したとき、あるいは、そのスレッドがI/O待ちなどでそれ以上実行を継続できなくなったときに、そのスレッドからCPUを取り上げ、スケジューリングを行い、次の実行時間（CPU割り当て時間）において実行すべきスレッドを選択する。

【0014】今、図13の状態でオペレーティングシステムが再スケジューリングして、次の実行時間に実行すべきスレッドを、実行可能なスレッドの候補の中から選択しようとしている場合を考える。

【0015】スレッド1～4は全て実行可能な状態であり、この他に直接これら4つのスレッドと関係の無いスレッドX、Y、Zがやはり同じように実行可能でスケジュールの対象になっているとする。なお、話を簡単にするためこれら7つのスレッドの実行優先度はみな等しいと仮定する。

【0016】この場合、オペレーティングシステムのスケジューラはどのスレッドを実行することもできる。しかし、次に実行すべきスレッドの選び方によっては、処理効率は大きく異なる。

【0017】まず、シングルプロセッサシステム（CPUが1つ）の場合を考える。タスク1について考えると、スレッド1、2、4が選択されても、これらはどれもロック待ちであるのでスピルループするだけである。しかし、スレッド3を選択して実行した場合は、4つの

スレッド全体の効率がよい。すなわち、スレッド3の処理が進めば、それだけ早く排他的な処理を終えることができ、ロックを解放し、他のスレッド1、2、4のいずれかがそのロックを確保して排他的な処理を実行することができるからである。

【0018】あるいは、ロックと関係ない（この時点でロック待ちになっていない）スレッドX、Y、Zを実行する場合も、CPUは有効に使用される。すなわち、この場合の結論としては、

10 10 【0019】1) ロック待ちのスレッド1、2、4（分類1）を選択するのはCPUの実行時間の浪費、2) ロックを確保しているスレッド3（分類2）を実行する場合は効率がよい、3) ロックとは関係の無いスレッドX、Y、Z（分類3）を実行する場合もCPUを有効に使える、ということがいえる。

【0020】しかしながら、従来の方法では、スケジューラはスレッドがロック待ちかどうか判断することができないので、分類1、2、3のようにスレッドを区別することができない。従って、通常は、スレッドの実行優先度だけでスレッドを選んでしまうので、必ずしも分類2、3に属すスレッドを選択することができず、ロック待ちのスレッドを選んでしまう場合が生じる。

20 20 【0021】図15には、このようにロック待ちのスレッドを選び効率悪くスレッドを実行する際のCPUの稼働状態の一例を示す。ここでは、ロックをかけて処理する部分の時間を実行処理単位時間（以降Tと表す）のほぼ1.5倍と仮定している。

【0022】時間t0からTの期間中においてスレッド3がロック状態である場合に、ロック待ちのスレッド1、2、4の順でCPUを割り当てるに、時間t1からt4までの期間は処理は実行されず、スピルループによってCPUが無駄に使用される。

【0023】そして、時間t4からの期間Tにおいてスレッド3のロック状態が解放されて初めて、スレッド1は、次の時間t5からの実行期間Tにおいてロックを確保でき、排他的に処理を行うことができる。この図15に示した範囲では、CPU使用率は半分以下であり、非常に効率が悪い。マルチプロセッサシステムの場合も同様の問題が生じる。やはりロック待ちのスレッドを選択すると無駄なスピルループでCPUを消費する。

30 30 【0024】図16には、CPU1～CPU4の4台のCPU構成のマルチプロセッサシステムにおいて、効率が悪いスレッド実行順序の一例が示されている。ロック待ちのスレッド1、2、4は、ロック状態のスレッド3の処理が進んでそのロックが解放されるまでスピルループしている。このため、この例においても、図に示した範囲でCPUの利用率は全体の約半分であり効率が悪い。

40 40 【0025】上記2つの例とも極端に悪い例である。しかし従来のUnixオペレーティングシステムのように

5

優先度に基づく多重レベルのラウンドロビンスケジューリングのみではこの様な現象が発生してしまい、システム全体が無駄なスピルループにより効率が悪化するという問題が生ずる。

【0026】なお、以上説明したスケジューリングの問題は、スレッド間での排他制御処理の場合のみならず、複数のプロセス間で共有メモリ上の同期変数を用いて排他制御を行う場合でも全く同様に生じるものである。

【0027】

【発明が解決しようとする課題】従来のスケジューリング方式では、スレッドやプロセス等のプログラム実行単位がロック待ち状態か否かに関係なくそれらの実行順を決定していたため、CPUの実行時間が浪費される欠点があった。

【0028】この発明はこのような点に鑑みてなされたものであり、スレッドまたはプロセスがロック待ち状態か否かに基づいてスケジューリングを実行できるようにし、CPUの利用効率を向上させることができるスケジューリング方式を提供することを目的とする。

【0029】

【課題を解決するための手段および作用】この発明のスケジューリング方式は、複数のプログラム実行単位間の相互排除を共有変数を用いて実現するコンピュータシステムにおいて、前記複数のプログラム実行単位それぞれがロック待ち状態にあるか否かを管理する手段と、ロック待ち状態でないプログラム実行単位に対して優先的にCPUが割り当てられるように前記複数のプログラム実行単位の処理順を決定するスケジューリング手段とを具備することを特徴とする。

【0030】このスケジューリング方式においては、複数のプログラム実行単位それぞれがロック待ち状態にあるか否かが管理され、実行候補のプログラム実行単位の中で、ロック待ち状態に無いプログラム実行単位に対して優先的にCPUが割り当てられるようにスケジューリングが行われる。このため、ロック待ちによってスピルループしている無駄な期間が少なくなり、CPUの利用効率を向上させることができる。

【0031】

【実施例】以下、図面を参照してこの発明の実施例を説明する。

【0032】まず、図1を参照して、この発明の一実施例に係わるスケジューリング方式の原理を説明する。オペレーティングシステムのカーネル10内に存在するスケジューラ100は、排他制御下で実行処理される複数のスレッドの処理順を決定するために、参照バス109を介してユーザ空間上のロック変数106～108を参照して次にCPUを割り当てる候補となった実行候補のスレッド（ここでは、スレッド101～105）がロック状態かロック待ち状態かを調べ、その結果を実際にC

6

PUをそのスレッドに割り当てるか否かの判断に使う。

【0033】このため、ロックを確保しているスレッド、または解除されたロック変数を待っていたスレッドを優先的に選択し実行することが可能となる。従って、従来のようにロック待ちのスレッドが頻繁に選択・実行され、CPUが無駄に使用される事を避けることができ、システム全体として効率の良いCPU割り当てが実現できる。図2には、スケジューラ100がロック変数を参照するための具体的な構成の一例が示されている。

【0034】この実施例では、新たに2つのデータ構造が用意されている。1つはタスク20の空間内の変数であり、ここでは「ステータス変数」と呼ぶ。もう1つはカーネル空間10内のデータであり、ここでは「ステータス変数ポインタ」と呼ぶ。これらは、いずれもポインタ変数である。

【0035】ステータス変数201、202はあらかじめスレッド101、102の空間に静的に宣言しており、スレッド101、102がロックを取れずロック待ち状態になる際にそのロック変数106のアドレス（L11）を格納する。また、ロック待ちでないとき、各スレッド101、102は自分のステータス変数201、202に“0”という値を入れておく。なお、ここでは、ロック変数106、109、110のアドレスは“0”以外の値になることが保証されていると仮定する。

【0036】ステータス変数ポインタ301、302は、生成されたスレッド101、102に対応してそれぞれ用意される。各スレッドは101、102は、あらかじめこのステータス変数ポインタ301、302のフィールドに、自分のステータス変数201、202のアドレス（A、B）をセットする。また、このための特別なシステムコールstatus-variable declare（）を用意しており、ステータス変数201、202のアドレス（A、B）を引数に指定し、このシステムコールを発行すればカーネル空間10内のステータス変数ポインタ301、302にそのアドレス（A、B）がセットされるようになっている。

【0037】この例における各スレッド101、102のステータス変数201、202の初期化、および、オペレーティングシステムのカーネル空間10に対する登録は、図3のフローチャートのように行われる。ここでは、C言語を用いて記述している。スレッドの数は10個とし、ステータス変数は10個用意するものとして、説明する。まず、long status[10]；

【0038】と配列でまとめて宣言し、ステータス変数[1]を“0”に初期化する（ステップS11、S12）。スレッド0のステータス変数のアドレスはstatus[0]、スレッド1はstatus[1]、…、スレッド9はstatus[9]、と対応する。次に、これらステータス変数のアドレスをシステムコール50 status-variable-declare

( ) でカーネル10に通知し、それぞれのステータス変数ポインタにアドレスをセットする (ステップS13)。

【0039】この結果、図4に示すように、10個のスレッド (スレッド0~9) と、そのスレッドのステータス変数のアドレス (status[0]~status[9]) を対応付けするテーブルがカーネル内に作られる。このテーブルは、図2のステータス変数ポインタ301, 302, …によって実現されるものにほかならない。

【0040】次に、図2の各スレッド101, 102がロック変数106をアクセスする際の手順を図5のフローチャートで説明する。また、図6には、C言語でコーディングした例が示されている。但し、図6では、アドレスL1のロック変数106はロックがかかっていない時、値OK (=0) を示し、また、関数t a s ( ) はt e s t & s e tを行う関数であり、戻り値がOKの時は引数に指定したロック変数が取得できたことを示している。

【0041】図5のフローチャートに示されているように、各スレッド101, 102は、アドレスL1のロック変数106の値からそのロック変数201がロックされているか否かを調べ (ステップS21)、ロックされてなければ、自分のステータス変数201, 202に値“0”を入れる (ステップS22)。次いで、スレッド101, 102は、t e s t & s e t命令等によってロック変数106のロック確保を試みる (ステップS23)。

【0042】ロック確保はいずれか1つのスレッドだけが可能であるので、ステップS24でロック確保を認識したスレッド、例えばスレッド101は、ロック状態になり排他的に処理を実行する。また、ステップS24でロック確保出来なかったスレッド、例えばスレッド102は、自分のステータス変数202にロック変数106のアドレスL1を入れ (ステップS25)、そして、ステップS21に戻ってスピンループする。次に、図7のフローチャートを参照して、スケジューラ100によるスケジューリングの際の処理手順を説明する。

【0043】まず、スケジューラ100は、通常のスケジューリング手法により、例えば待ち行列にキューイングされている複数のスレッドの中から予め定められた優先度にしたがって最も優先順位の高いスレッドを候補として1つ選ぶ (ステップS31)。

【0044】次に、スケジューラ100は、カーネル空間10内にあるそのスレッドのステータス変数ポインタを参照して、そのスレッドに対応するステータス変数ポインタが登録されているか否かを調べる (ステップS32)。もし、登録されていなければ、ロック変数を利用した実行許可判断が不要であると認識して、そのスレッドにCPUを割り当てる (ステップS38)。一方、候

補として選ばれたスレッドに対応するステータス変数ポインタが登録されているれば、ロック変数を利用した実行許可判断を行うために、ステップS33~S37の処理に進む。

【0045】ここでは、スケジューラ100は、ステップS31で実行候補として選んだスレッドに対応するステータス変数ポインタを参照することによってそのスレッドに対応するステータス変数のアドレスを求め (この際必要に応じて、アドレス値の正当性のチェック、論理アドレスから物理アドレスへの変換、あるいは二次記憶装置から主記憶へのページの転送、等を行い)、実行候補のスレッドのステータス変数に登録されている値を読み取る (ステップS33)。

【0046】このときステータス変数の値が“0”ならば (ステップS34)、その実行候補のスレッドはロック待ちの状態でないことが分かるので、スケジューラ100はステップS38に進んでCPUをそのスレッドに割り当てる実行する。

【0047】また、もしステータス変数の値が“0”でないときは (ステップS34)、このスレッドは前回実行時にロック待ちであったことを意味している。この場合は、ステータス変数の値はそのスレッドが待っているロック変数のアドレスなので、スケジューラ100は、そのスレッドが属するタスク空間内のロック変数を直接参照し (この際も必要に応じて、アドレス値の正当性のチェック、論理アドレスから物理アドレスへの変換、あるいは二次記憶装置から主記憶へのページの転送、等を行い)、ロックが解除されているかどうかを判断する (ステップS35, 36)。

【0048】この時点でロックが解除されなければ (ロック変数=“0”)、スケジューラ100はステップS38に進んでCPUをそのスレッドに割り当てる実行する。一方、ロックが解除されていなければ (ロック変数=“1”)、このスレッドの実行を見送り、次の候補として他のスレッドを選択する (ステップS37)。この手順を実行スレッドが決まるまで繰り返す。

【0049】このようにして、ロック変数の値を参照してスケジューリングを行うことにより、ロック待ちのプロセスを選び無駄にCPU時間を消費することを避けることができる。

【0050】図8、図9には、このような手順でスケジューリングを行った場合のCPUの稼働状況の一例が示されている。図8はシングルプロセッサシステムの場合に相当し、また図9は4台のCPUから成るマルチプロセッサシステムの場合に相当するものである。

【0051】また、これら図8、図9では、図13で前述したようにスレッド3がロック状態であり、スレッド1, 2, 4がロック待ち状態で、さらにスレッドX, Y, Zがロックに関係しない場合を想定している。

【0052】図8の例では、時間t0からTの期間中に

においてスレッド3がロック状態である場合に、そのロック状態のスレッド3に対して優先的に次の実行期間（時間t0からTの期間）にCPUが割り当てられ、その期間Tにおいてスレッド3のロックが解除されると、そのロック解除を待っていたスレッド1にCPUが割り当てられる。図9のマルチプロセッサシステムの場合も同様にして、ロック状態のスレッドS3、またはロックに関係しないスレッドY、Y、Zが優先的に実行される。

【0053】このようにロック待ちでないスレッドにCPUを割り当てることによって、CPU利用率は100パーセントになり、無駄なスピンループを大幅に減少することができる。

【0054】以上のように、この実施例においては、スケジューラ100がスケジューリングの際に、必要に応じて実行候補のスレッドに対応するロック変数を参照し、そのスレッドがすぐにロックを確保できるかどうかを調べ、CPUを割り当てるか否かの判断の参考とする。この結果、ロック待ち状態にないスレッド、つまり解除されたロック変数を待っていたスレッド、またはロック状態のスレッド、を優先的に選んで実行できるので、スピンループによるCPUタイムの無駄な消費が少なくなり、効率よい処理が実現できる。

【0055】尚、ここでは、複数のスレッド間の相互排除を共有変数を用いて実現するシステムについてのみ説明したが、この発明によるスケジューリングは、図12に示したような共有メモリを持ち複数のプロセス間の相互排除を共有変数を用いて実現するシステムについても同様に適用できる。

【0056】また、この実施例では、図9に示すように、ロックされているロック変数を持っている状態のスレッドは選択していない。しかし、マルチプロセッサシステムにおいて、プロセッサ数が多く、スケジューリング時にプロセッサが余っている場合や、ロックをかけているスレッドがすでに実行中である（従って比較的早くロックが解除される見込みがある）場合には、ロック解除されていないロック変数を持つスレッドにCPUを割り当てるようなスケジューリング方針も有効である。この様なスケジューリングも本発明を適用することにより実現することができる。

【0057】

【発明の効果】以上述べたように、この発明によれば、スレッドまたはプロセスがロック待ち状態か否かに基づいてスケジューリングを実行できるようになり、CPUの利用効率を向上させることができる。

【図面の簡単な説明】

【図1】この発明の一実施例に係わるスケジューリング方式の原理を示すブロック図。

【図2】同実施例の具体的構成の一例を示すブロック図。

【図3】同実施例におけるステータス変数の初期化／登録動作を説明するフローチャート。

【図4】同実施例におけるスケジューラで管理されるスレッドとステータス変数との対応関係を示すテーブルを示す図。

【図5】同実施例におけるロック確保のための動作を説明するフローチャート。

【図6】図5のフローチャートをC言語で記述した際のプログラム文を示す図。

【図7】同実施例におけるスケジューリング動作を説明するフローチャート。

【図8】同実施例におけるスケジューリング動作をシングルプロセッサシステムで実行した際のCPUの稼働状況を示す図。

【図9】同実施例におけるスケジューリング動作をマルチプロセッサシステムで実行した際のCPUの稼働状況を示す図。

【図10】通常のプロセスのメモリ空間を示す図。

【図11】通常のスレッドのメモリ空間を示す図。

【図12】通常の共有メモリを持つプロセスのメモリ空間を示す図。

【図13】従来のスケジューリング方式を説明するためのブロック図。

【図14】従来のスケジューリング方式が適用されるシステム内での排他制御の動作手順を説明するフローチャート。

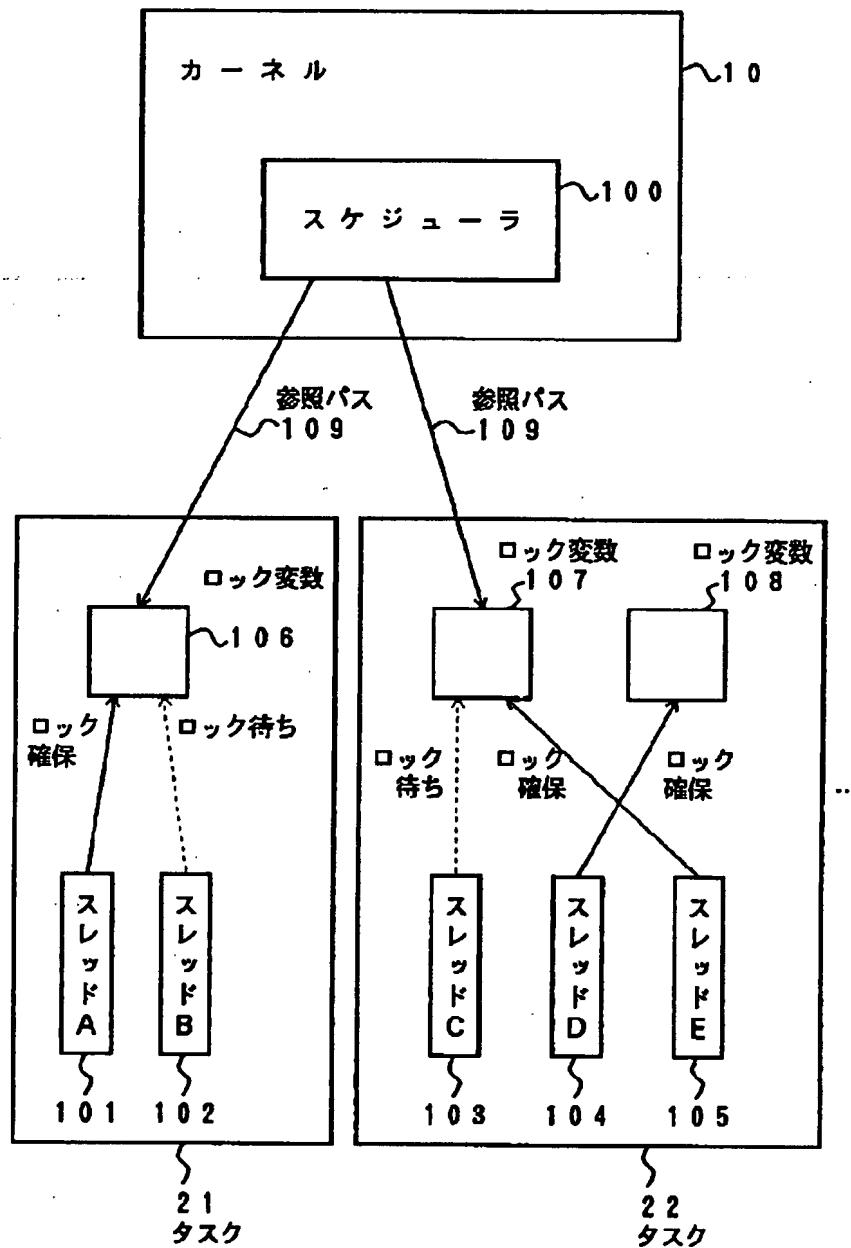
【図15】従来のスケジューリング方式をシングルプロセッサシステムで実行した際のCPUの稼働状況を示す図。

【図16】従来のスケジューリング方式をマルチプロセッサシステムで実行した際のCPUの稼働状況を示す図。

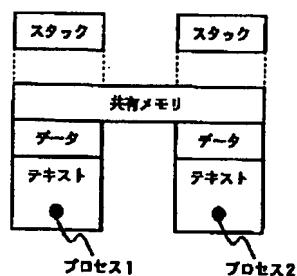
【符号の説明】

10…カーネル空間、21, 22…タスク、100…スケジューラ、101～105…スレッド、106～108, 109, 110…ロック変数、201, 202…ステータス変数、301, 302…ステータス変数ポインタ。

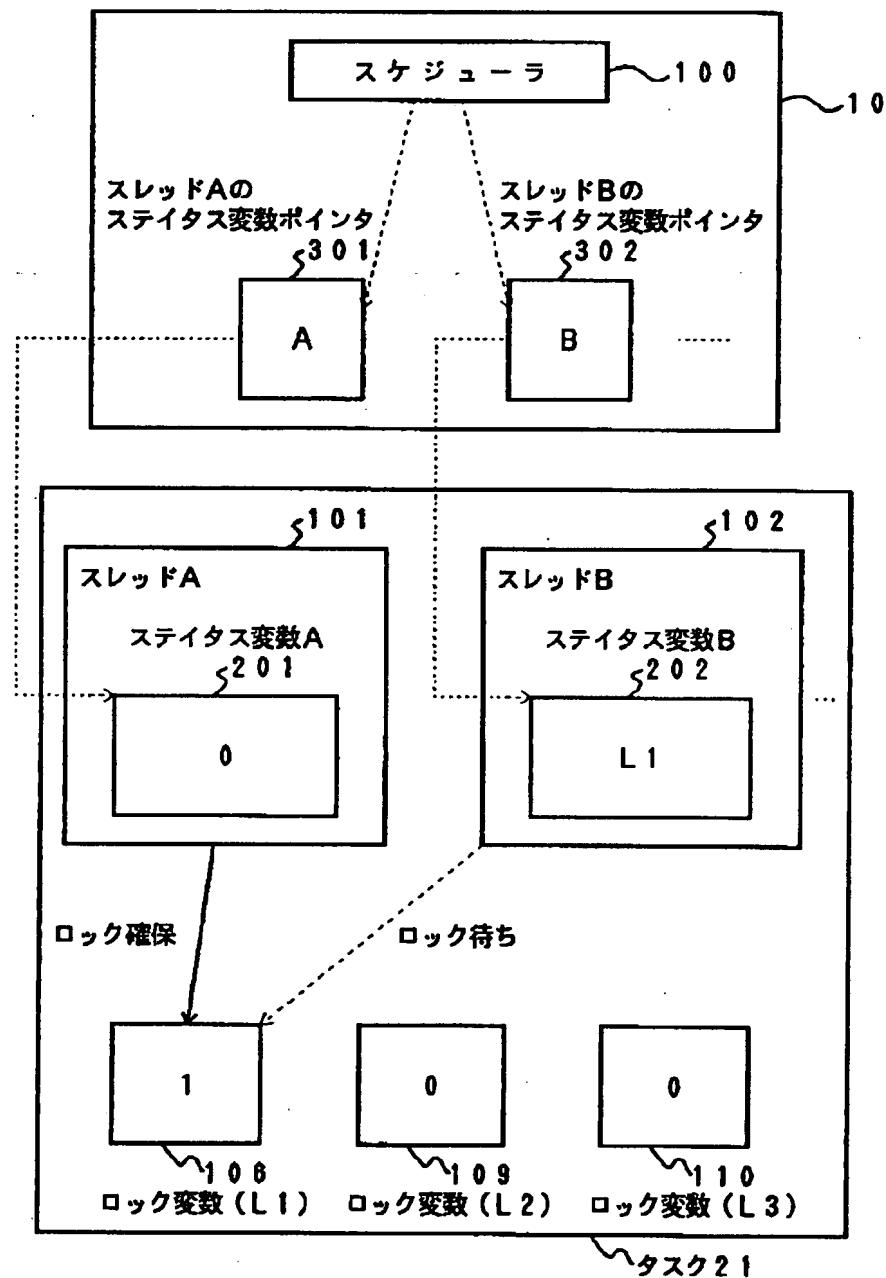
【図1】



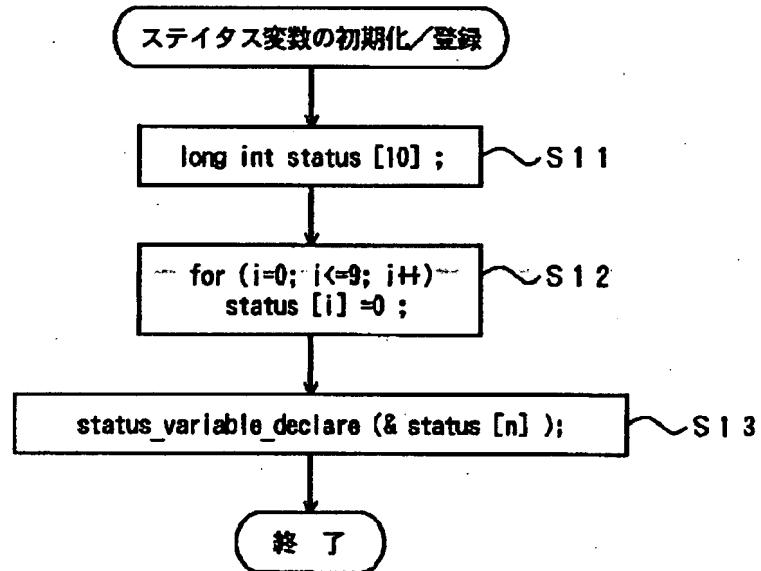
【図1 2】



【図2】



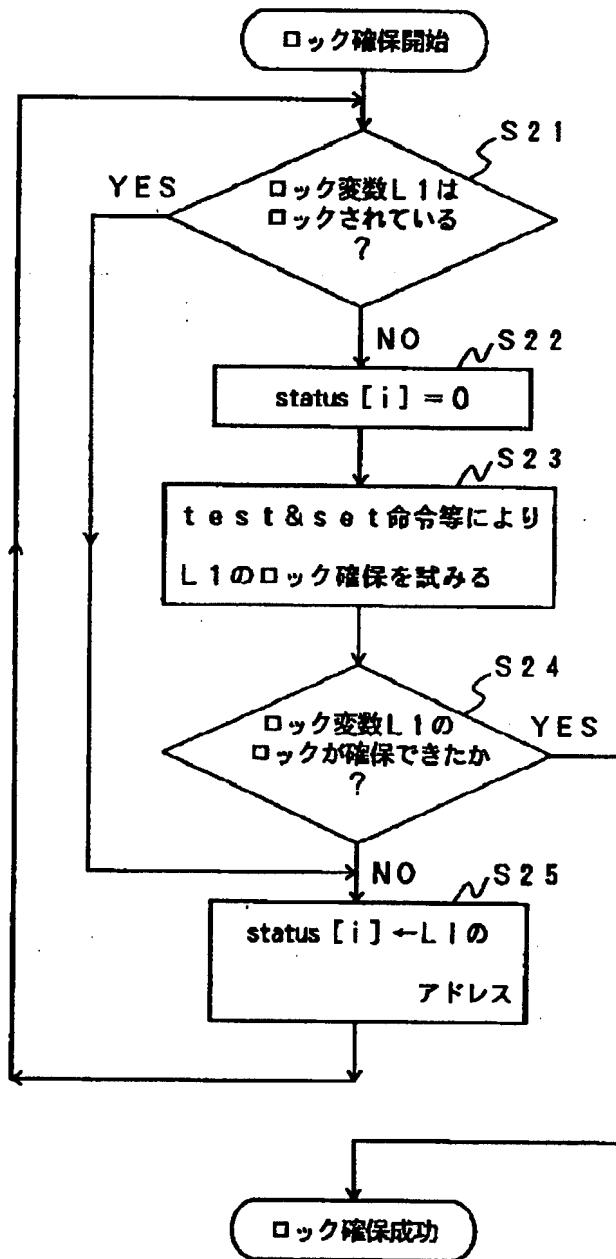
【図3】



【図4】

スレッド名	ステータス変数名
スレッド0	status [0]
スレッド1	status [1]
⋮	⋮
スレッド9	status [9]

【図5】



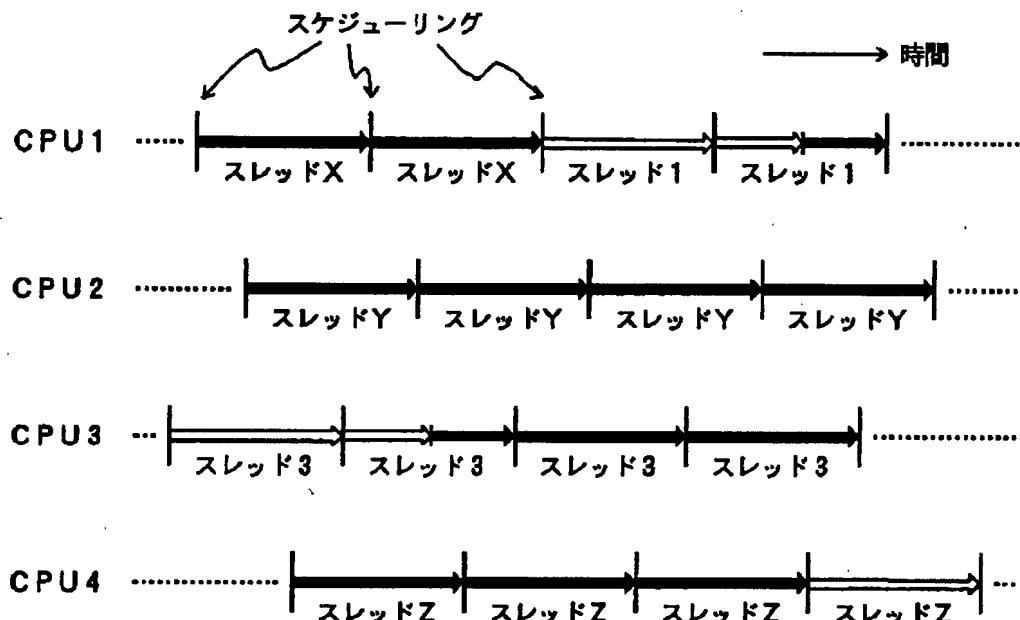
【図6】

```

while(1) {
    if(L1==OK) {
        status[i]=0;
        if(tas(&L1)==OK)
            break;
    }
    status[i]=&L1;
}

```

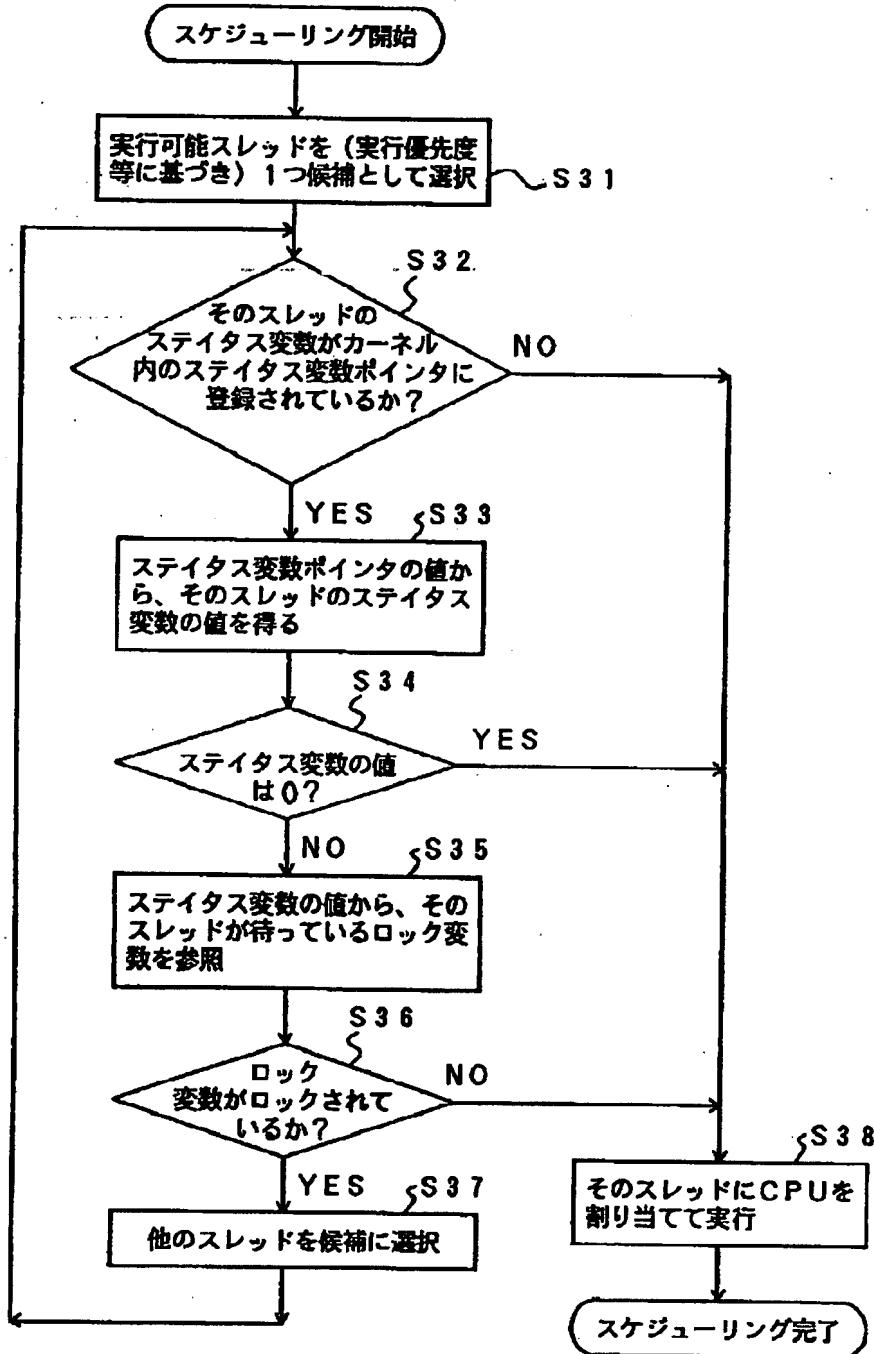
【図9】



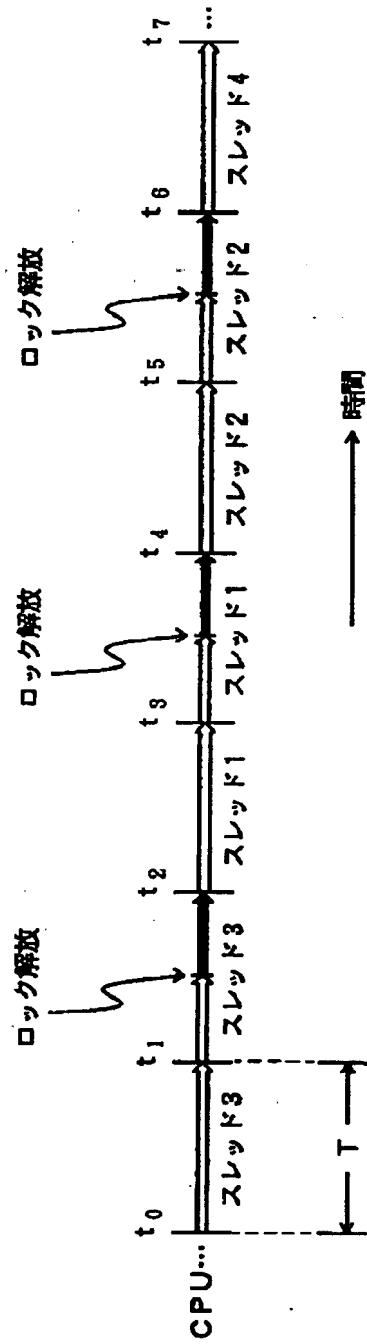
⇨ : ロックを確保し、排他的に処理をしている部分

→ : ロックを必要としない処理の部分

【図7】



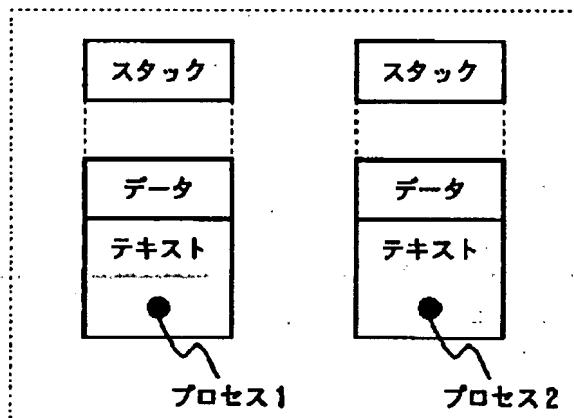
【図8】



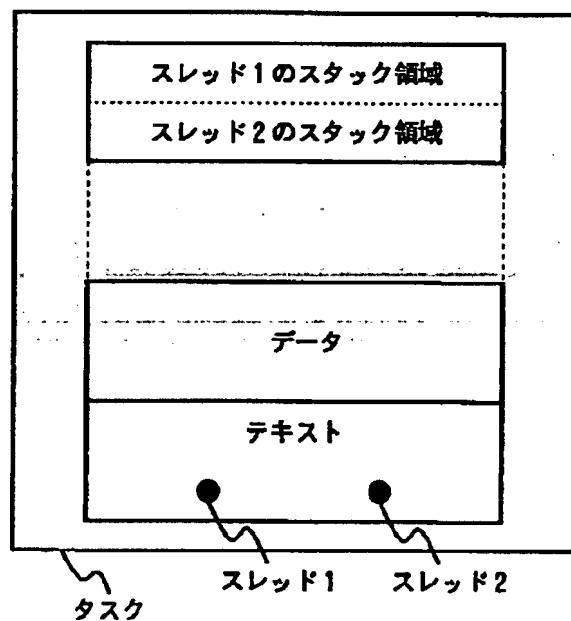
□: ロックを確保し、排他的に処理をしている部分

▶: ロックを必要としない処理の部分

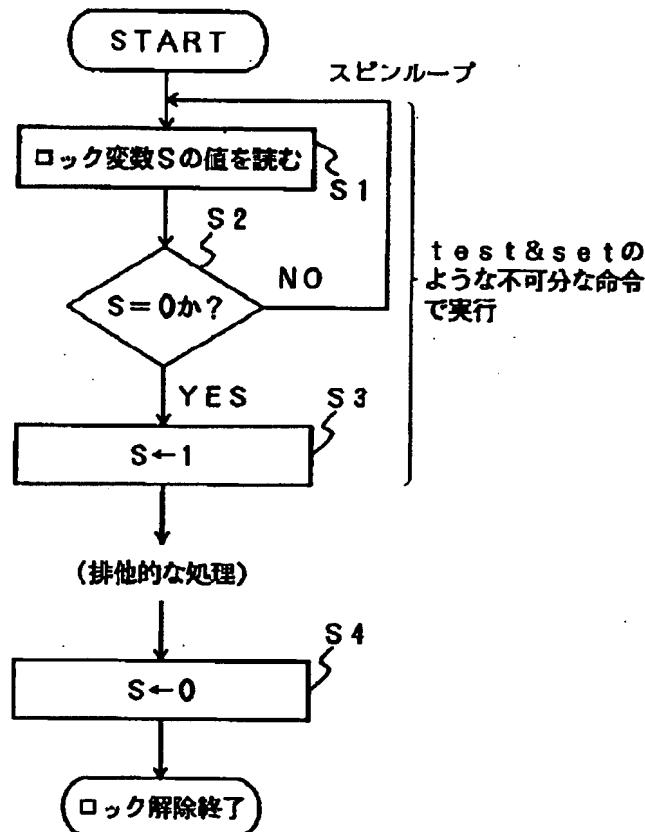
【図10】



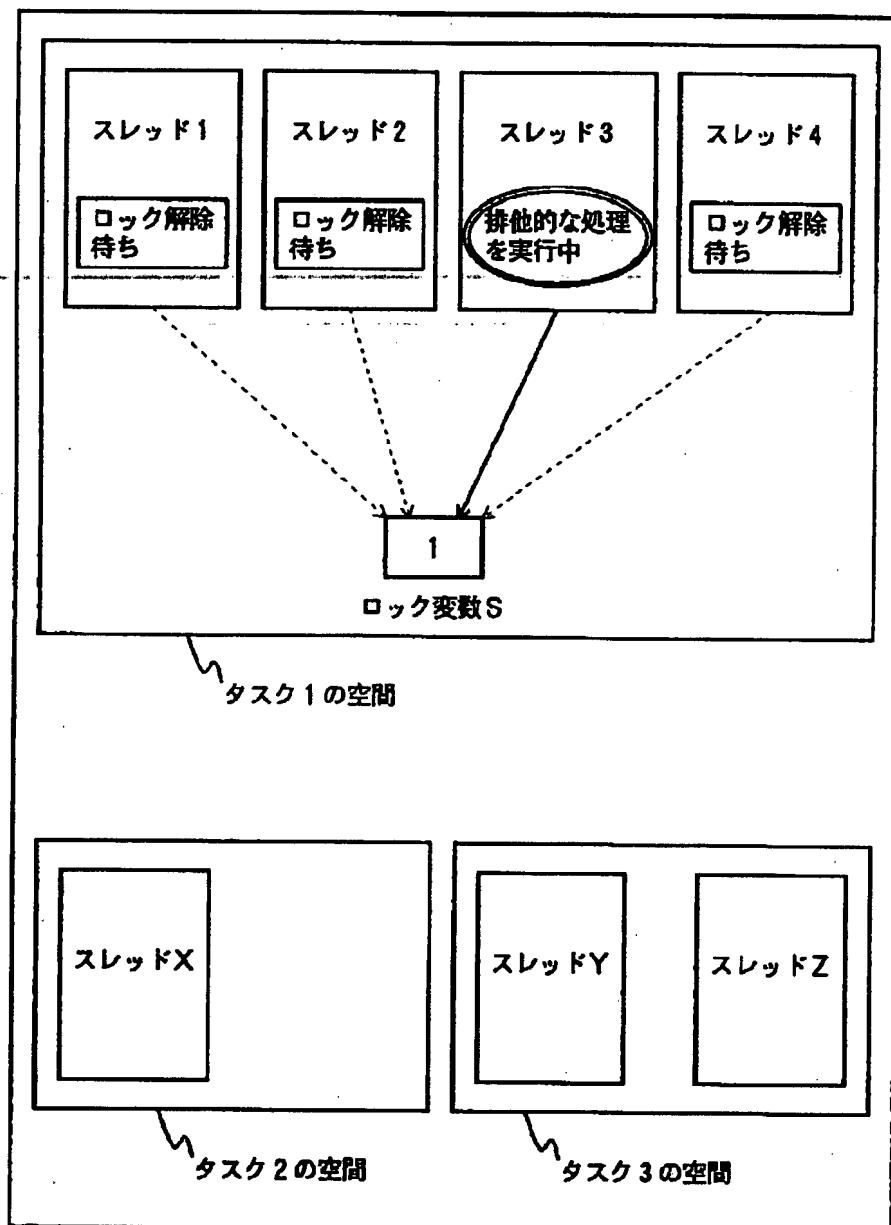
【図11】



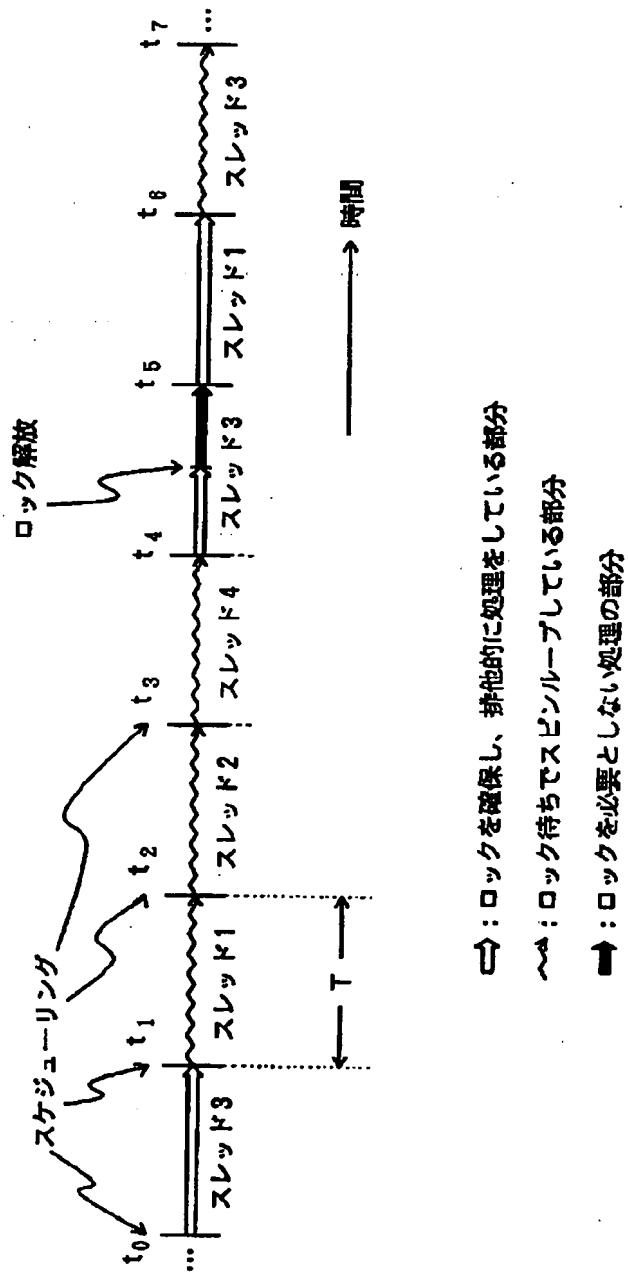
【図14】



【図13】



【図15】

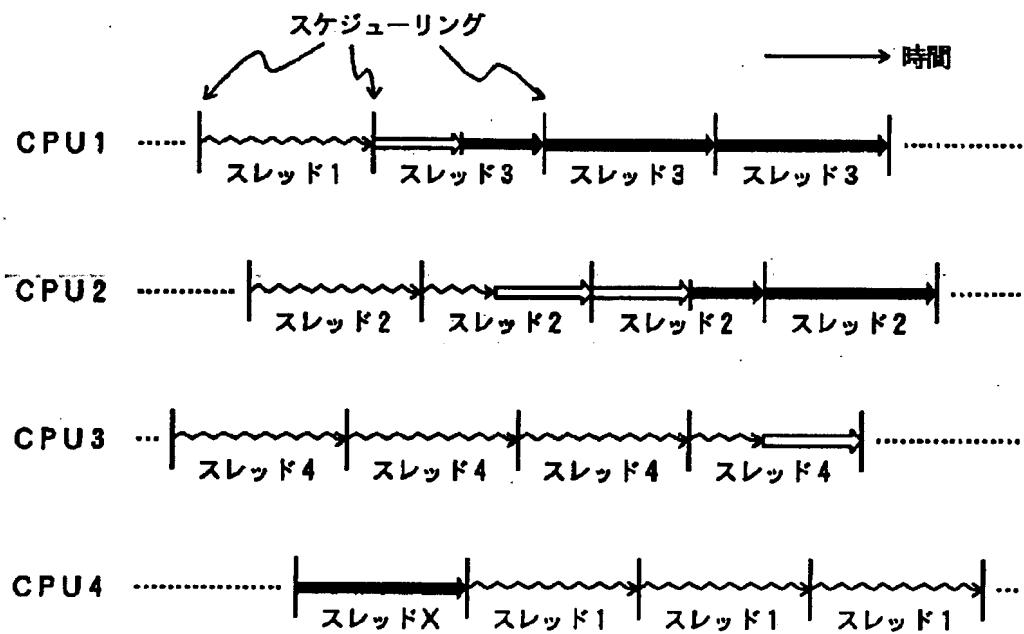


⇨: ロックを確保し、排他的に処理をしている部分

~~: ロック待ちでスピンループしている部分

→: ロックを必要としない処理の部分

【図16】



⇒ : ロックを確保し、排他的に処理をしている部分

～～ : ロック待ちでスピンループしている部分

→ : ロックを必要としない処理の部分